HW2 Documentation

Itamar Barron – 208981159

Roy Frumkis – 312472129

Introduction

This document explains the implementation details of HW2. HW2 is a program that receives a command file and a number of threads to be used to process it. The command file contains a list of jobs, where each job is composed of one or more basic commands. The program creates a specified number of threads and assigns jobs from the command file to these threads with job_queue. Each thread processes its assigned jobs and updates counters stored in separate files on disk. Additionally, the program can log events to log files if log enabled is set to 1 and also displaying statistics to an additional file it creates.

Course of action

The commands file is being read by the code. In it, dispatcher is responsible for analyzing the command line arguments that are given and making sure that the number of parameters given is correct. It creates a specified number of counter files and worker threads. Each counter file is called countxx.txt, where xx is a number between 0 to 99 and will be writing with the job lines the specific thread, the one that initialized as the xx thread number, will execute. For the dispatcher commands we have created functions that immediately performs the desired action and for the job lines that are consist of basic commands we have wrote functions that insert them into the job queue and performs them by order with available threads. We only create worker threads during initialization, and they will be alive for the entire run of the program; if a thread has no work to do at some point, it goes to sleep until new work is available to pick up. After finishing processing the command line file and executing the whole commands in it and in the job queue, the relevant threads are being killed and the additional files are updating. That's when we are finishing the run of the code.

Functions and helpers

We have implemented the code by the orders we have received and to do that we have used functions and helpers. The most significant ones will be clarified here. For the documentation to be clear as possible we will organize it as the additional C files are organized in the submission folder.

basic_commands.h + basic_commands.c

Includes the basic commands we have asked to create.

- **msleep(int x):** suspends the execution of the calling thread for x milliseconds.
- **increment(int x):** reads a file named "countxx.txt" where "xx" is the number x passed as an argument. It reads a long long integer from the file, increments it, and writes it back to the file.
- **decrement(int x):** reads a file named "countxx.txt" where "xx" is the number x passed as an argument. It reads a long long integer from the file, decrements it, and writes it back to the file.
- **run_job(void *_commands_str):** runs a sequence of commands passed as a string argument. It first creates an array of commands and arguments by parsing the string, then executes each command in order. If a command is "msleep", it calls the msleep() function with the argument. If a command is "increment", it calls the increment() function with the argument. If a command is "decrement", it calls the decrement() function with the argument. If a command is "repeat", it repeats the sequence of commands that follow for a specified number of times, as indicated by the argument. The function prints the executed commands to the console.

job_queue.h

This header file defines the structures and functions used in a job queue.
The **Job struct** contains a function pointer and an argument pointer, which will be used to pass a function to a worker thread, and an optional argument to the function. It also contains start and end times, which are used to measure the running time of a job.
The **Archive struct** contains a linked list of Job objects that have been completed, along with a count of the number of jobs in the list.
The **Queue struct** contains pointers to the head and tail of the queue, along with a mutex and two condition variables used to synchronize access to the queue. It also contains an Archive object. The job struct contains the general start time to be used for threads logging.
The **ThreadData struct** is used to pass data to worker threads, including a pointer to the Queue object, a flag indicating whether a log file should be created, and a thread number.
The functions defined in this header file allow the creation and manipulation of a job queue, including adding jobs to the queue, removing jobs from the queue, creating worker threads, and printing statistics

about completed jobs. The header file also contains function prototypes for functions that are implemented in other source files.

job_queue.c

This code defines functions and data structures for a job queue and worker threads that execute jobs from the queue.

The Queue struct defines a job queue and contains pointers to the head and tail of the queue, as well as a mutex and two condition variables used for synchronization. The Archive struct is used to keep track of completed jobs.

- **create_queue(struct timeval start_time):** creates a new job queue and initializes the mutex and condition variables associated with it. Also receives start time for later time calulations for threads logs.
- **enqueue(Queue *queue, Job *job):** adds a job to the end of the job queue, signals the condition variable cond_q_non_empty to wake up a waiting worker thread, and releases the lock on the mutex associated with the job queue.
- **dequeue(Queue *queue):** removes and returns the first job from the job queue. If the job queue is empty, the function waits on the cond_q_non_empty condition variable until a job is added to the queue. The function also moves the dequeued job to an archive linked list, and updates the job count in the archive.
- **worker_thread(void *arg):** this is the function executed by each worker thread. It waits for a job to be added to the job queue, dequeues the job, and executes the function associated with the job. It also logs the start and end times of the job if the create_log flag is set.
- **kill(void *arg):** this is a special job function that is used to signal worker threads to exit. It takes no argument and does nothing.
- **print_archive(Archive *archive):** prints the job archive linked list, which stores all jobs that have been dequeued from the job queue. Used for debug.
- **print_job_stats(Archive *archive):** calculates and prints statistics on the job turnaround times, including the sum, minimum, maximum, and average turnaround times.
- **print_job_stats_to_file(Archive *archive, FILE *file):** prints the same job statistics as print_job_stats(), but writes them to a file instead of printing them to the console.
- **add_cmnd_job(Queue *queue, char *cmnd):** creates a new job with the run_job function and the specified argument cmnd, and adds it to the job queue.
- **add_kill_job(Queue *queue):** creates a special job with the kill function and adds it to the job queue to signal the worker threads to exit.
- **free_queue(Queue *queue):** frees all jobs in the job archive, the mutex and condition variables associated with the job queue, and the job queue itself.
- **pthread_create_wrapper(pthread_t *thread, Queue *queue ,int create_log , int thread_num):** a wrapper function that creates a new worker thread and initializes a ThreadData struct with the specified arguments. This function is used instead of pthread_create() in order to create a log file for each worker thread.

- **create_thread_data(Queue *queue, int create_log, int thread_num):** creates a new ThreadData struct with the specified arguments.
- **free_thread_data(ThreadData *data):** frees the ThreadData struct.
- **wait_for_queue_empty(Queue *queue):** waits until the job queue is empty. This function is used to block the main thread until all jobs have been completed.

main.h + main.c

This files containes the running functions of the code. There will be set the order of the run and the terms of operation and in it will be created, if needed, the extra .txt files and the statistics that will be added to every run of the code.

- **is_not_worker(char* first_term):** a function that receives a pointer to a character array (string) and checks if the first 6 characters of the string are "worker". If the first 6 characters are not "worker", it returns 1 (indicating that it is not a worker), otherwise, it returns 0 (indicating that it is a worker).
- **create_counter_files(int num_counters):** a function that receives an integer value num_counters, which represents the number of counter files to create. The function generates a file name with a counter value in the range of 0 to num_counters-1 and creates the file. The contents of the file are initialized to 0.
- **create_threads(pthread_t* thread_ptrs, struct Queue* queue, int num_threads, int save_logs):** a function that receives a pointer to an array of pthread_t structures, a pointer to a Queue structure, an integer value num_threads (number of threads to create), and an integer value save_logs (if 1, logs will be saved to a file, otherwise, not). The function creates num_threads worker threads and stores their pointers in the thread_ptrs array. The threads are initialized with the worker_thread function and the provided queue as an argument.
- **count_worker_lines(FILE* fp):** a function that receives a pointer to a FILE structure representing the input file. The function reads the file line by line and checks if the first 6 characters of the line are "worker". If it is, it increments the count of worker lines. The function returns the count of worker lines.
- **dispatcher_command(char* line, int save_logs, struct Queue *queue):** a function that receives a pointer to a character array (string) representing a dispatcher command, an integer value save_logs (if 1, logs will be saved to a file, otherwise, not), and a pointer to a Queue structure. The function extracts the command and its argument from the input string and executes it. If the command is "msleep", the function sleeps for the provided time. If the command is "wait", the function waits for all pending background commands to complete before continuing to

process the next line in the input file. If an unknown command is encountered, the function prints an error message. After executing the command, the function writes a log entry to the dispatcher.txt file if save_logs is 1.

- **handle_command(char* line, int save_logs, Queue *queue):** a function that receives a pointer to a character array (string) representing a command, an integer value save_logs (if 1, logs will be saved to a file, otherwise, not), and a pointer to a Queue structure. The function checks if the command is for the dispatcher or a worker thread by checking the first 6 characters of the line. If it is a dispatcher command, the function calls the dispatcher_command function. Otherwise, it extracts the clean worker job and adds it to the provided queue.
- **create_stats_file():** a function that creates and returns a pointer to a FILE structure representing the stats.txt file. If the file cannot be created, the function prints an error message and returns NULL.
- **write_stats_file(long long total_elapsed_time, FILE *file):** a function that receives a long long value total_elapsed_time (the total elapsed time in milliseconds) and a pointer to a FILE structure representing the stats.txt file. The function writes the total elapsed time to the stats file.
- **main(int argc, char *argv[]):** the main function of the program. It receives the command-line arguments, opens the input file, parses the arguments, creates the counter files, creates the worker threads, reads the input file line by line, handles each command, sends kill jobs to the queue, waits for all threads to finish, calculates the total elapsed time, creates and writes a stats file that includes the total elapsed time and job statistics, prints the job statistics to the console, frees the queue and its resources, and finally returns 0. The program follows a dispatcher-worker model, where the main thread acts as a dispatcher and enqueues jobs to a queue. The worker threads dequeue jobs from the queue and execute them. The program also supports logging, which can be enabled by passing a command-line argument. Overall, the main function orchestrates the entire program's execution, from parsing inputs to creating threads, executing jobs, collecting statistics, and freeing resources.

In summary, this project aimed to develop a job dispatcher that receives commands from an input file, and dispatches them to worker threads for execution. The main program receives command-line arguments, creates the necessary resources, and manages the worker threads and job queue. The worker threads execute the jobs in parallel and update a job archive that is used to generate statistics about the execution. The project successfully achieved its goals and provides an efficient and scalable solution for job dispatching. Through this project, we gained valuable experience in implementing multi-threaded applications, memory management, and file I/O.